**docker container run nome-imagem** > faz download da imagem não encontradas, criação do container, execução do container, faz uso do container no modo interativo.

**docker container stop** > Parar serviço de um container

**docker container ps** > Mostrar containers ativos.

**docker container ps -a** > Mostrar containers independente do status.

**--rm** > remove containers

**exit** > sair do container

**docker container run -it nome\_container bash** > criar novo container, o -it significa que o container será criado no modo iterativo e que teremos acesso ao terminal do próprio container. Após criado o container, teremos acesso ao seu terminal.

**touch** > cria arquivo dentro do container

**ls** > listar conteúdos

**docker container run - -name nome-para-renomear -it nome-original bash** > cria container e renomeia seu nome

**docker container ls** > listar containers

**docker container start -ai nome-container** > startar determinado container no modo iterativo e permitir acesso ao terminal desse container, -ai é equivalente ao -it.

**curl** > obtem conteúdo de um link

**cd pasta/** > entrar em determinada pasta

**cd ..** > sair de uma pasta

**mkdir diretório** > criar pasta ou diretório

**Isolamentos**

**Mapeamento de portas**

**docker container run -p 8080:80 imagem** > mapear porta, número antes dos dois pontos é porta referente ao host e o número depois dos dois pontos é porta referente ao container.

**Mapeamento de volumes (pastas)**

Primeira maneira:

Entrar em um diretório e depois usar esse comando:

**docker container run -p 8080:80 -v $(pdw)/html:/usr/share/nginx/html nome\_imagem** > Para mapear volume use esse comando. Use a flag **-p**

com a porta (externa) do host que são os números antes dos **:** e com a porta (interna**)** do container que são os números depois do **:**, assim você mapeia primeiro as portas. Use a flag **-v** com o comando **$(pwd)** que irá executar todo o comando dentro da pasta corrente que entramos antes de digitar todo esse comando, depois do **$(pwd)** será digitada a sub-pasta **/html** que está dentro da pasta corrente, a sub-pasta **/html** que será mapeada dentro do container, o conteúdo antes dos **:** pega a pasta do host que será mapeada pelo container e o conteúdo depois dos **:** é o caminho da pasta que será feito o mapeamento dentro do container. O comando **$(pwd)** não funciona no windows, mas a alternativa é do modo a seguir.

**docker container run -p 8080:80 -v "C:/Users/lufon/Desktop/docker/ex-volume/html:/usr/share/nginx/html" nome\_imagem** > Esse comando faz exatamente o que o anterior faz, mas ao invés de usar o comando **$(pwd)/html** colocamos o caminho absoluto até a pasta que será mapeada. Esse comando funciona no windows.

**Rodar containers em modo deamon (background)**

Rodar o container em modo deamon ou em background significa que o container vai ficar executando. Um modo de executar quando o projeto está em desenvolvimento. Aqui que o Docker começa a brilhar de verdade.

Exemplos:

**docker container run -d - -name nome\_para\_renomear -p 8080:80 -v $(pdw)/html:/usr/share/nginx/html imagem** > O que realmente faz o container ser executado em modo daemon é a flag **-d**. Pode fazer desse jeito, com várias opções junto ou somente executar em modo daemon. Desse jeito está executando o container em modo daemon, renomeando o nome da imagem, mapeando portas, mapeando volume. Nesse caso com o mapeamento de volumes com o comando **$(pwd)** não funcionários no windows, então pode-se substituir o trecho **$(pwd)/pasta\_mapear** pelo caminho absoluto da pasta **html** desse modo:

**docker container run -d --name ex-daemon-basic -p 8080:80 -v "C:/Users/lufon/Desktop/docker/ex-volume/html:/usr/share/nginx/html" imagem**

**Gerenciando containers**

**docker container start id ou nome container** > Startar o container

**docker container restart id ou nome container** > Reiniciar o container

**docker container stop id ou nome container** > Parar o container

**docker container ps** > Verificar containers ativos

**docker container ps -a** > Verificar containers independente do status

**Manipulação de containers em modo daemon**

**docker container ls**, **docker container list**, **docker container ps** > Ambos comandos verificam containers ativos. Adicionando a flag **-a** na frente desses comandos, conseguimos mostrar todos os containers independente do status.

**docker container logs id ou nome container** > Mostrar os logs feitos por um container.

**docker container inspect id ou nome container** > Mostrar informações do container.

**docker container exec id ou nome container uname -or** > Mostrar que tipo de sistema está sendo executado dentro do container.

**docker image ls** > Listar imagens.

**docker image rm flag nome-imagem** > Remover imagens. Se colocar a flag **-f**, **--force** força a remoção da imagem. Se colocar a flag **–no-prune**, não exclui sem pais.

**docker volume ls** > Listar volumes.

**docker volume rm flag nome-volume** > Remover volumes. Se colocar a flag **-f**, **--force** força a exclusão de um ou mais volumes.

**Deixando de ser apenas um usuário**

**Container** > O container é um processo isolado da máquina host.

**Imagem** > E um modelo de sistema de arquivos com várias camadas que a partir dessas camadas é criado o container.

A imagem é como a classe na poo e o container é como um objeto, o container nasce da imagem como um objeto é instanciado a partir de uma classe. A partir de uma classe podemos instanciar um ou vários objetos e seguindo esse paradigma uma imagem pode nascer um ou vários containers.

**docker container - -help** > mostrar todos os sub-comandos relativos aos containers.

**docker image - -help** > mostrar todos os sub-comandos relativos as imagens.

**docker volume - -help** > mostrar todos os sub-comandos relativos aos volumes.

**Entendendo mais as imagens**

Toda imagem (bem como os containers) possuem um identificador único em formato hash usando sha256. Porém seu uso não é muito prático, então para simplificar isto o docker utiliza uma tag para identificar imagens.

A tag normalmente é formada por um nome, seguido de : dois pontos e depois uma versão. É extremamente comum utilizar uma versão chamada latest para representar a versão mais atual.

Exemplos de tags de imagens:

• nginx:latest

• redis:3.2

• redis:3

• postgres:9.5 17 Ζ

Na prática uma tag é apenas um ponteiro para o hash da imagem, e várias tags podem apontar para o mesmo hash.

**Comandos básicos no gerenciamento de imagens**

Já usamos de maneira implícita o recurso de download de imagens docker, agora vamos entender melhor o gerenciamento de imagens.

**docker image pull** **<tag>** - Baixa a imagem solicitada, este comando pode ser executado implicitamente, quando o docker precisa de uma imagem para outra operação e não consegue localiza-la no cache local.

**docker image ls** - Lista todas as imagens já existentes.

**docker image rm** **<tag>** - Remove uma imagem do cache local.

**docker image inspect** **<tag>** - Extrai diversas informações utilizando um formato JSON da imagem indicada.

**docker image tag** **<source> <tag>** - Cria uma nova tag baseada em uma tag anterior ou hash.

**docker image build -t** **<tag>** - Permite a criação de uma nova imagem.

**docker image push <tag>** - Permite o envio de uma imagem ou tag local para um registry.

**Docker Hub x Docker Registry**

**Docker Registry** - É uma aplicação server side para guardar e distribuir imagens Docker.

**Docker Hub** - É um serviço de registro de imagens Docker em nuvem, que permite a associação com repositórios para build automatizado de imagens. Imagens marcadas como **oficiais** no Docker Hub, são criadas pela própria **Docker Inc**. E o código fonte pode ser encontrado em: <https://github.com/docker-library>

- A linha de comando possui o comando **docker search** para procurar imagens no Docker Hub.

**Criação de imagens**

**Primeiro build**

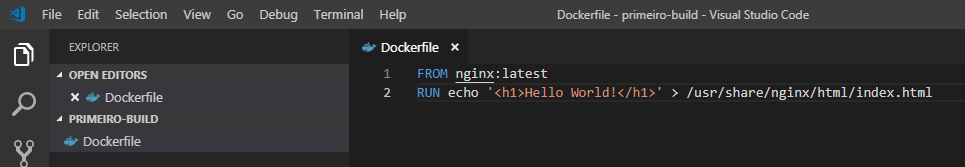
O processo consiste em criar uma imagem a partir de um arquivo de instruções. Criamos um arquivo de texto chamado **Dockerfile** em um diretório e configuramos alguns comandos que são como instruções de como será criada a imagem. Posteriormente, no terminal executaremos o comando **docker build** para criar a imagem a partir do Dockerfile.

Depois de criar a imagem executamos o comando **docker container run -p porta-host:porta-container nome-dado-para-a-imagem .** para criar um container a partir da imagem criada, para chegar vá no navegador, digite [**http://localhost:porta-host**](http://localhost:porta-host).

Detalhadamente explicando...

**1 –** Crie um diretório e crie dentro dele um arquivo e renomeie com exatamente o nome **Dockerfile** com f minúsculo e sem extensão de arquivo.

**2 –** No **Dockerfile** para uma configuração bem básica, informe o comando **FROM** com a tag de uma imagem já criada como por exemplo **nginx:versão** para que a imagem que estamos criando seja criada a partir dela, informe também o comando **RUN** passando como argumento **echo ‘<h1>Hello World!</h1>’ > /usr/share/nginx/html/index.html**. O RUN vai fazer com que o echo imprima **<h1>Hello World!</h1>** no arquivo index.html dentro do diretório do nginx.



Fizemos um exemplo bem básico do uso do Dockerfile, mas é possível fazer muito mais configurações, mais a frente aprenderemos mais sobre comandos dentro do Dockerfile.

**3 –** Abra o terminal, navegue com o comando **cd** até o diretório onde está o Dockerfilee crie uma imagem com o comando **docker image build -t** desta maneira:

**docker image build -t nome-imagem .**

O docker image build significa que será criada uma imagem, o -t significa que a seguir terá o nome ou tag da imagem e o ponto no final significa que será usado o diretório local que é até onde navegamos com o comando cd e que onde está o Dockerfile que a imagem será criada a partir dele.

**4 –** Com o comando **docker image ls** verifique se a imagem foi criada.

**5 –** Para criar um container a partir da imagem que criamos anteriormente digite o comando:

**docker container run -p 8080:80 nome-imagem-criada-anteriormente**

O docker container run irá criar o container, como a imagem será criada a partir da imagem do nginx que é um servidor então iremos mapear portas para o container utilizar, mas se não fosse um servidor o mapeamento dessas portas seria opcional. A flag **-p** significa que será feito mapeamento de portas, o número antes dos **:** significa a porta do host (externa) que será utilizada para acessar o container, o número depois dos **:** será a porta do container (interna) e por ultimo será informado o nome ou a tag da imagem criada anteriormente.

**6 –** Para testar vá no navegador, digite http://localhost e dois pontos, digite a porta externa:

Exemplo:

[**http://localhost:8080**](http://localhost:8080)

**Instruções para a preparação da imagem**

Outras instruções para se usar no Dockerfile.

**FROM** - Especifica a imagem base a ser utilizada pela nova imagem.

**LABEL** - Especifica vários metadados para a imagem como o mantenedor. A especificação do mantenedor era feita usando a instrução específica, **MAINTAINER** que foi substituída pelo **LABEL**.

**ENV** - Especifica variáveis de ambiente a serem utilizadas durante o build.

**ARG** - Define argumentos que poderão ser informados ao build através do parâmetro **--build-arg**.

**Exemplo do Dockerfile com passagem de parâmetros e criação de imagens e containers passando parâmetros**

DockerFile:

**FROM** debian

**LABEL** maintainer ‘Lucas Barbosa Fonseca <lufonsecabarbosa95@gmail.com’

**ARG** S3\_BUCKET=files

**ENV** S3\_BUCKET=${S3\_BUCKET}

O **ARG** define que um argumento será passado na hora que ocorrer o build, ou seja, no momento em que a imagem for criada com o comando docker image build. Esse é o argumento que iremos receber. Esse argumento está com o valor padrão que é files, mas pode ter outros.

Com o **ENV** você pode aplicar em cima do ARG variáveis de ambiente.

Criando image passando novo parâmetro para o ARG:

**docker image build - -build-arg S3\_BUCKET=myapp -t nome-imagem .**

Criando container a partir da imagem criada anteriormente e mostrando argumento através do comando bash que foi passado para o ARG:

**docker container run nome-imagem-criada-anteriormente bash -c ‘echo $S3\_BUCKET’**

Consultando informações da imagem usando filtros para trazer somente a informação desejada, por exemplo o maintainer da imagem:

**Docker image inspect - -format=”{{index .Config.Labels \”maintainer\”}}” nome-imagem**

**Instruções para povoamento da imagem**

**COPY** - Copia arquivos e diretórios para dentro da imagem.

**ADD** - Similar ao anterior, mas com suporte extendido a **URLs**. Somente deve ser usado nos casos que a instrução COPY não atenda.

**RUN** - Executa ações/comandos durante o build dentro da imagem.

**Exemplo do Dockerfile com povoamento nas imagens**

Dockerfile:

**FROM** nginx:1.13

**LABEL** maintainer 'Lucas Barbosa Fonseca <lufonsecabarbosa95@gmail.com>'

**RUN** echo '<h1>Sem conteúdo</h1>' > /usr/share/nginx/html/conteudo.html

**COPY** \*.html /usr/share/nginx/html/

O **RUN** vai executar o comando echo que vai imprimir um h1 no arquivo **conteudo.html** dentro do diretório **/usr/share/nginx/html/** que é o diretório do servidor nginx.

Para rodar o arquivo  **conteudo.html** , precisamos copiar com o comando **COPY** o **index.html** para dentro do diretório do nginx **/usr/share/nginx/html/**, pois o **conteudo.html** está dentro do **index.html.**

**COPY** \*.html /usr/share/nginx/html/

O **COPY** vai copiar o arquivo, o **\*** vai selecionar todos os arquivos html que estão no diretório local, ou seja, o diretório que o Dockerfile se encontra. O **/usr/share/nginx/html/** é o diretório do nginx onde o arquivo **index.html** vai ser colocado, o \* está selecionando o arquivo **index.html**.

**Instruções com configurações para execução do container**

**EXPOSE** - Informa ao Docker que a imagem expõe determinadas portas remapeadas no container. A exposição da porta não é obrigatória a partir do uso do recurso de redes internas do Docker. Recurso que veremos em Coordenando múltiplos containers. Porém a exposição não só ajuda a documentar como permite o mapeamento rápido através do parâmetro -P do docker container run.

**WORKDIR** - Indica o diretório em que o processo principal será executado.

**ENTRYPOINT** - Especifica o processo inicial do container.

**CMD** - Indica parâmetros para o **ENTRYPOINT**.

**USER** - Especifica qual o usuário que será usado para execução do processo no container (**ENTRYPOINT** e **CMD**) e instruções **RUN** durante o build.

**VOLUME** - Instrui a execução do container a criar um volume para um diretório indicado e copia todo o conteúdo do diretório na imagem para o volume criado. Isto simplificará no futuro, processos de compartilhamento destes dados para backup por exemplo.

Exemplo prático dessas instruções:

Nesse exemplo prático criaremos um servidor na linguagem de programação python com o nome **run.py** e esse servidor vai ser responsável por servir a nossa página **index.html**.

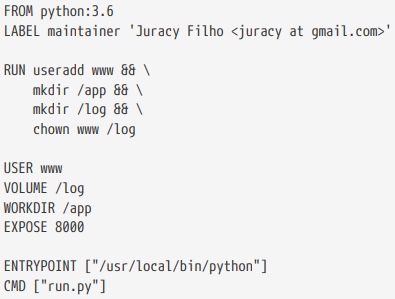
(O servidor é mais para fazemos esse exemplo prático, porém não vamos dar muita ênfase para esse servidor)

Primeiramente criaremos a página **index.html** com um <p> básico e criaremos o servidor python **run.py** com o seguinte código:



Em seguida iremos criar o arquivo **Dockerfile** dentro de um diretório que será o mesmo onde vamos criar o arquivo **index.html** e o **servidor python run.py**:

Dockerfile:



Através do descritor **Dockerfile** será criada uma imagem e essa imagem conterá o servidor **python run.py**  que por sua vez vai servir o arquivo **index.html**.

Explicando as instruções dentro do Dockerfile:

O **FROM** indica a imagem que a imagem que vamos criar vai se basear. O **LABEL** indica o mantenedor da imagem que iremos criar. O **RUN** irá executar os comandos **username** que vai criar o usuário www, o comando **mkdir** que vai criar o diretório /app, o comando **mkdir** novamente para criar a o diretório /log e o comando **chown** diz que o dono do diretório /log é o usuário www. Nesse exemplo de Dockerfile, esses comandos dentro do RUN serão executados dentro do bash do container, os **&&** dentro do RUN irão concatenar os comandos e a **/** é para quebrar as linhas, quando for executar vários comandos dentro do RUN é interessante colocar todos dentro de um mesmo RUN. O **USER** indica o usuário que vai estar logado quando os processos **ENTRYPOINT**, **CMD** e **RUN** forem executados no container durante o build. O **VOLUME** cria um volume para um diretório indicado que no caso é /log, e será copiado todo o conteúdo que está dentro do /log na imagem para dentro do volume. O **WORKDIR** indica o diretório em que o processo principal será executado. O **EXPOSE** indica a porta que será exposta através da imagem que será criada. No momento que for executar o container podemos usar a porta indicada nesse comando ou pode remapear no comando docker container run quando for executar o container e usar a porta remapeada. O **ENTRYPOINT** indica o processo inicial que será executado no container, é o que será executado primeiro no arquivo, nesse exemplo será executado o python:3.6 então nesse comando será informado o diretório onde a imagem do python:3.6 está instalada. E por fim o **CMD** indica parametros para o comando **ENTRYPOINT**, o comando ENTRYPOINT vai chamar a imagem do python e o CMD vai incrementar o ENTRYPOINT executando o run.py que é o servidor.

Criando imagem a partir do Dockerfile:

Para criar uma imagem a partir do arquivo descritor Dockerfile, no terminal com o comando **cd** vamos navegar até a pasta onde está localizado o Dockerfile e digitar o seguinte comando:

**docker image build -t nome\_imagem .**

Esse comando vai criar uma imagem a partir do Dockerfile.

Criando container a partir da imagem criada recentemente:

Para criar um container a partir da imagem criada recentemente, depois de digitar o comando para criar a imagem e executar, digite no terminal o seguinte comando:

**docker container run -it -v $(pwd):/app -p 80:8000 - -name nome-container nome-imagem-criada-recentemente**

Dessa maneira acima, fazendo mapeamento de volume com o comando **$(pwd)** não funciona no windows. Como uma alternativa podemos usar o caminho absoluto da pasta que queremos mapear, dessa forma:

**docker container run -it -v “C:/Users/lufon/Desktop/docker/build-dev:/app” -p 80:8000 - -name nome-container nome-imagem-criada-recentemente**

O docker container run irá criar um container, o **-it** indica que o container será criado no modo interativo e que teremos acesso ao terminal do container, o **-v** indica que a seguir será feito um mapeamento de volume, vai pegar todo o conteúdo da pasta da máquina host e vai jogar dentro da pasta que está no container, assim o container vai ler o arquivo run.py que está dentro da pasta na máquina host. O **-p** indica que a seguir ocorrerá um mapeamento de portas, a porta 8000 será remapeada para a porta 80, o **- -name** indica que iremos renomear o nome padrão do container por um nome de nossa escolha e por ultimo vamos colocar o nome da imagem que criamos anteriormente, pois o container será criado a partir dessa imagem.

**Subir imagens para o repositório do Docker Hub**

Primeiramente para verificar qual imagem queremos subir para o repositório do Docker Hub, vamos listar todas as nossas imagens digitando o comando a seguir: **docker image ls**

Para subir a imagem é importante criar uma nova imagem a partir da imagem escolhida para subir, mas essa nova imagem será criada com uma tag diferente, para isso vamos usar o comando **docker image tag**, por exemplo:

**docker image tag** **nome\_antigo nome\_novo**

No **nome\_antigo** vamos colocar o nome da imagem a qual a nova imagem será criada e no **nome\_novo** vamos colocar o nome de usuário da nossa conta do Docker Hub, em seguida uma / e por ultimo vamos colocar um nome, dois pontos e a versão da nossa imagem, por exemplo:

**docker image tag ex-simple-build lufonseca/simple-build:1.0**

Antes de subir a imagem vamos fazer o login no Docker Hub no terminal usando o comando **docker login**. Na frente da flag **- -username** vamos informar nosso nome de usuário da nossa conta do Docker Hub. A senha será pedida após executar o comando **docker login**.

**docker login - -username=lufonseca**

Finalmente depois de se logar, vamos subir a imagem com o comando **docker image push**, nesse comando só iremos informar o nome da imagem que queremos subir.

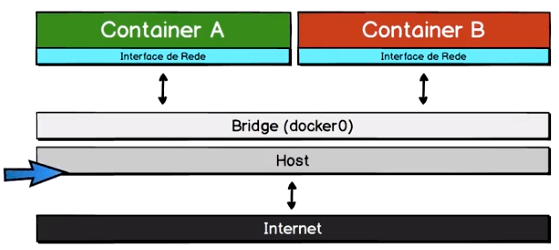
**docker image push** **lufonseca/simple-build:1.0**

**Redes no Docker**

No Docker podemos segregar algumas redes para que possamos ter um ambiente mais seguro escondendo serviços de outros serviços dentro de nossa estrutura ou expondo ao mundo exterior o realmente precisa ser exposto.

No docker temos 4 tipos de rede mas vamos nos concentrar em 3, que são:

**Bridge Network:** O modelo bridge vem por padrão em todos os containers que são criados no docker. O bridge funciona como uma ponte entre a rede dos containers e a rede do host da máquina fazendo uma espécie de isolamento. Os containers no modo bridge possuem uma interface de rede própria com seus próprios endereços ip. Quando criamos containers no modo bridge, por padrão esses containers pertencem a mesma rede e se comunicam entre si, mas podemos criar outra rede bridge separada, com endereço ip diferente e mesmo que um container estiver em uma rede e outro container em outra rede podemos criar um mecanismo para a rede desses containers conseguirem se comunicar. Também é possível criar novas redes.



Usando:

**docker network ls** – Listar redes existentes.

Executando o comando acima, será mostrada várias informações dentre elas terá uma informação chamada **Subnet** que informa um ip, por exemplo **127.17.0.1**. Esse ip é dividido entre ip da rede do container e o ip do container dessa rede. Os dois primeiros elementos são o ip da rede e os dois ultimos elementos são o ip do container dessa rede. O ip pode chegar até **127.17.255.255**.

**docker container run -d - -name nome\_container nome\_imagem sleep 1000** – O container será criado. O -d significa que o container será criado no modo daemon, o - -name significa que o container será renomeado com um nome de sua escolha, depois de informado o nome do container vamos informar o nome da imagem a qual o container irá se basear e o sleep 1000 significa que o container ficará ativo e executando.

**docker container exec -it nome\_container ifconfig** – Mostrar toda a interface de rede do container.

**docker container exec -it nome\_container1 ping ip\_rede\_e\_container2** – Testar conectividade de um container com outro. Para testar a conectividade entre containers primeiro executamos o comando no modo iterativo usando a flag -it, depois informamos o nome do container o qual queremos saber se está conectado com outro, depois ping para ver se está pingando e informamos o ip de rede e do container o qual o container 1 quer se conectar. Para a conexão der certo os containers devem estar na mesma rede.

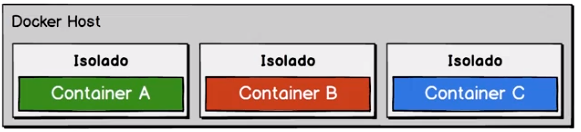
**docker network create - -driver nome\_driver\_rede nome\_rede** – Criar uma nova rede a partir de um driver de rede. O docker network create significa que uma rede será criada, a flag - -driver significa que a rede será criada a partir de um driver de rede, depois informamos o driver de rede que ´pode ser none, bridge ou null e por ultimo informamos o nome da nossa rede. Quando criamos uma rede um novo ip é criado, se o ip de uma ultima rede criada for 127.17.0.0 então se criaremos outra rede o ip vai ser 127.18.0.0.

**docker network inspect nome\_rede** - Inspecionar rede.

**docker network connect nome\_rede nome\_container** – Conectar um container a uma rede. Primeiro informamos o nome da rede o qual queremos que o container se conecte e por ultimo informamos o nome do container que queremos conectar a rede.

**docker network disconnect nome\_rede nome\_container** - Desconectar um container de uma rede. Primeiro informamos o nome da rede o qual queremos que o container se desconecte e por ultimo informamos o nome do container que queremos desconectar da rede.

**None Network:** O modelo none não permite que os containers tenham acesso a rede e por seu vez ter acesso a internet e também não permite acesso de um container a outro. Podemos usar esse modelo no caso de nossos containers só fazerem processos internos como mapear volumes.



Usando:

**docker container run -d - -net none imagem\_container** – Esse comando cria um container com o modelo nome e no modo daemon. O -d significa que o container será criado no modo deamon e o - -net none significa que o container vai ser criado com o modelo nono.

**docker network ls** – Mostrar todas as redes no docker.

**docker container run - -rm - -net none alpine ash -c “ifconfig”** - Como no exemplo mostrado no video, esse comando cria um container e remove logo em seguida e mostra a interface de rede do container registrada. O comando - -rm significa que o container será removido após a criação, o - -net nome significa que o container será criado com o modelo none, alpine é uma distribuição do linux que será a imagem do container, o ash é o um bash do linux que é mais leve e o comando -c “ifconfig” será executado em cima do ash, o -c “ifconfig” mostra a interface de rede do container registrada;

**Host Network:** O modelo host faz com que a interface de rede dos containers tenham acesso direto a rede do host sem nenhuma ponte ou isolamento. Com esse modelo temos uma proteção muito baixa e tudo fica exposto, mas por outro lado oferece muita performance. O uso desse modelo só é recomendado no caso de o projeto demandar muita performance.

**docker container run - -rm alpine ash -c “ifconfig”**

**docker container run - -rm - -net none alpine ash -c “ifconfig”**

**docker-compose up -d - -scale nome\_container\_docker\_compose=qtd\_instancias**

**docker-compose logs -f -t nome\_container\_docker\_compose**

**docker-compose exec container psql -U postgres -d banco\_de\_dados -c ‘select \* from emails’**